|  |
| --- |
| Detector  项目文档 |
|  |
| 董玮豪 |

目录

[1. 概述 5](#_Toc20321259)

[1.1 架构 5](#_Toc20321260)

[1.2 输入 6](#_Toc20321261)

[1.3 数据处理 8](#_Toc20321262)

[1.4 检测 8](#_Toc20321263)

[1.4.1 标签 8](#_Toc20321264)

[1.4.2 标签传递 8](#_Toc20321265)

[1.4.3 恶意状态 10](#_Toc20321266)

[1.5 数据持久化 10](#_Toc20321267)

[2. 功能模块介绍 10](#_Toc20321268)

[2.1 Main 11](#_Toc20321269)

[ApiLCSDetectorMain.java 11](#_Toc20321270)

[OfflineMain.java 11](#_Toc20321271)

[OfflineMainLocalJson.java 11](#_Toc20321272)

[2.2 KafkaProcessor 11](#_Toc20321273)

[ReceiverManager.java 11](#_Toc20321274)

[MessageReceiver.java 12](#_Toc20321275)

[MessageProcessor.java 12](#_Toc20321276)

[TopicKey.java 16](#_Toc20321277)

[2.3 ETWproto 16](#_Toc20321278)

[2.4 EventRecord 16](#_Toc20321279)

[EtwEventType.java 16](#_Toc20321280)

[EventFormat.java 16](#_Toc20321281)

[EventRecord.java 17](#_Toc20321282)

[ProtobufParsedMessage.java 17](#_Toc20321283)

[2.5 ExtraDetectionFactors 17](#_Toc20321284)

[Certificate.java 17](#_Toc20321285)

[Macro.java 17](#_Toc20321286)

[NetworkActivity.java 17](#_Toc20321287)

[PowerShell.java 17](#_Toc20321288)

[ProcessInjection.java 18](#_Toc20321289)

[SensitiveFile.java 19](#_Toc20321290)

[SensitiveProgram.java 19](#_Toc20321291)

[VisibleWindow.java 19](#_Toc20321292)

[2.6 Process 20](#_Toc20321293)

[Process.java & ProcessTable.java 20](#_Toc20321294)

[计算ProcessName 20](#_Toc20321295)

[计算uuid 21](#_Toc20321296)

[2.7 Framework 22](#_Toc20321297)

[Label/LabelType.java 22](#_Toc20321298)

[Label/LabelForCS.java 23](#_Toc20321299)

[Event/EventType.java 23](#_Toc20321300)

[Event/EventForCS.java 23](#_Toc20321301)

[Event/NetworkEventForCS.java 23](#_Toc20321302)

[ISignalForCS.java 24](#_Toc20321303)

[Grammar 25](#_Toc20321304)

[Node/Node.java 25](#_Toc20321305)

[Node/ProcessNode.java 25](#_Toc20321306)

[Node/FileNode.java 25](#_Toc20321307)

[DetectionFramework.java 26](#_Toc20321308)

[2.8 Neo4j 28](#_Toc20321309)

[Task/BaseTask.java 29](#_Toc20321310)

[Task/AddFileNodeTask.java 29](#_Toc20321311)

[Task/AddProcessNodeTask.java 29](#_Toc20321312)

[Task/AddLabelTask.java 29](#_Toc20321313)

[Task/AddNodeRelationTask.java 29](#_Toc20321314)

[Task/ChangeNameTask.java 30](#_Toc20321315)

[Task/DeleteNodeRelationTask.java 30](#_Toc20321316)

[Task/UpdateProcessNodeTask.java 30](#_Toc20321317)

[Neo4jClient.java 30](#_Toc20321318)

[2.9 CrossMachine 31](#_Toc20321319)

[IntranetSocket.java 31](#_Toc20321320)

[ProcessNetworkEvent 32](#_Toc20321321)

[2.10 Illustrator 33](#_Toc20321322)

[node/BaseNode.java 33](#_Toc20321323)

[node/FileNode.java 33](#_Toc20321324)

[node/ProcessNode.java 33](#_Toc20321325)

[edge/BaseEdge.java 33](#_Toc20321326)

[edge/IncomingEdge.java 33](#_Toc20321327)

[edge/OutgoingEdge.java 34](#_Toc20321328)

[NodeWrapper.java 34](#_Toc20321329)

[Link.java 35](#_Toc20321330)

[Common.java 35](#_Toc20321331)

[DrawAttackGraph.java 35](#_Toc20321332)

# 1. 概述

## 1.1 架构

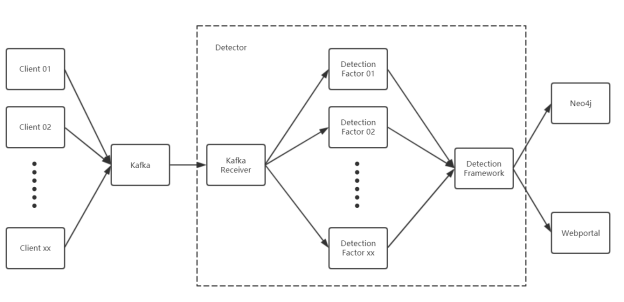


图1 架构图

上图是本系统的架构图。每个用户机器上运行一个data collector，采集检测需要的审计数据并发送到kafka。数据发送前会经过ProtoBuf序列化和Snappy压缩。虚线框中的部分是detector。Kafka Receiver从kafka中拉取数据并进行处理，再根据数据类型不同发往各个不同的检测模块。Detection Factors为不同的检测模块，会从审计数据中提取语义生成标签或事件，再传递给Detection Framework。Detection Framework储存所有进程和文件的状态，并根据传入的标签和事件驱动标签的传递。当framework检测出某个攻击，会将相关信息发送到Web展示界面（Webportal）。同时framework会异步地将其状态备份到neo4j中。当某个进程结束或某个文件被删除时，会被framework从内存中删除，但仍然存在与neo4j中，用于以后的溯源。

## 1.2 输入

detector从kafka接收来自collector的数据。一条数据为一个event，一个event有一个主体，一个事件类型，和一个客体。比如对于进程A写文件B这个事件，主体为进程A，事件类型为写，客体为文件B。

一条事件有以下字段：

processId：该事件主体的进程id

threadId：主体的线程id

eventName：事件类型

timestamp：事件的时间戳，除以1000000后为System.currentTimeMillis()的格式。

callstack：在正常事件中为null，在PHF（Potential Harmful Function，即Remotedesktop，Keylogger等恶意行为）事件中为PHF的名字。

arguments：该事件的参数，如FileIoRead事件中，参数为被读的文件名。每种事件类型有自己特定的参数列表。

collector端每凑齐10000条数据会发送一次，按照正常的思路，数据结构应该是这样：

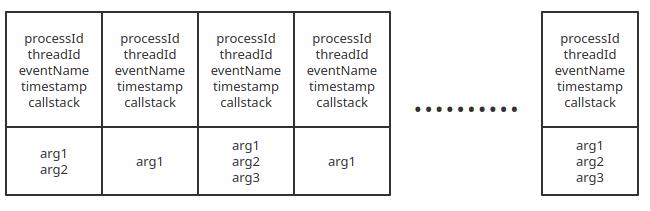


图2 正常思路下的数据结构

但由于每种事件的参数数量不同，使用这种方式进行protobuf序列化的时候性能会比较差，因此我们使用了下面这种方式：

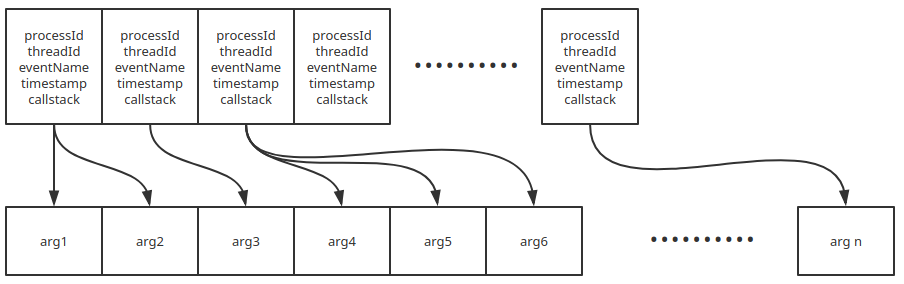
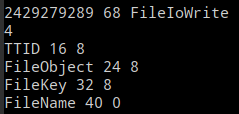


图3 实际采用的数据结构

将参数展开成，单独作为一个一维数组发送。但这样我们就很难判断哪个参数对应哪个事件了。于是我们引入了format文件。

format.txt是发送端和接收端之间的协议，约定了某种事件有哪些参数。举个例子，下图是format文件里面的一项，它的含义是，FileIoWrite事件有4个参数，分别是TTID，FileObject，FileKey和FileName。参数后面跟的两个数字对接收端没有作用，可以忽略。



发送端在发送时并不会以String的形式发送eventName，而是用ProviderId和Opcode组合成一个Unique Key来表示eventName。上图中，2429279289是ProviderId，68是Opcode。这两个数字在一起表示FileIoWrite类型。

NOTE1：使用这一手段进行数据传送时，如果根据format算出来的argument总数和实际收到的argument数量不同，说明要么是collector端和detector端的format不一致，但很难查出来具体的不一致点。排查方法将在下文中介绍。

## 1.3 数据处理

收集到一条事件后，我们会根据它的事件类型，将其传入不同的检测模块。比如对于文件读写事件，会将其送入SensitiveFileAccess模块，检测该进程是否访问了敏感文件。如果有，我们会为其创建一个标签，表示它访问了敏感文件，并送入检测模块。

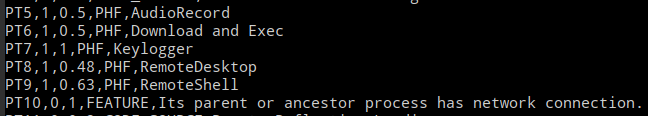
我们也会选取有意义的事件送入检测模块，比如文件读事件，文件写事件，进程启动事件等，都会被送入检测模块。

## 1.4 检测

### 1.4.1 标签

我们采用了基于状态的检测算法。首先我们定义了一些标签（<https://docs.qq.com/sheet/DSmRRbEVnR2NRU2dB?tab=BB08J2>）（写在label.txt中），在用于描述进程/文件的当前状态。比如当某个进程带有Process Transmit Label 1（PT1）时，说明它有网络连接。当某个文件带有File Transmit Label 1（FT1）时，说明它包含来自网络的数据。

下图展示了label.txt的一部分：



这是一个csv格式的文件，第一列是标签编号，最后一列是标签描述，中间三列不用管。

### 1.4.2 标签传递

进程与文件之间的事件会导致状态的传递，比如当一个带有网络连接的进程写了一个文件，我们就认为该文件包含了来自网络的数据，如下图所示：

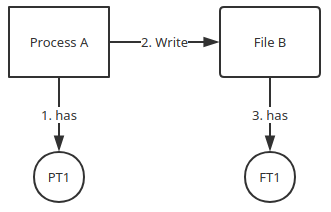
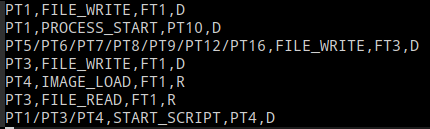


图4 一个标签传递的例子

通过我们收集的数据无法判断A是否真的向B写入了来自网络的数据，因此我们只能通过这种粗粒度的规则来进行猜测。这些规则写在trans.txt。

下图展示了trans.txt的一部分：



这是一个csv格式的文件，第一列是事件主体标签L1，第二列是事件类型E，第三列是事件客体标签L2，第四列是传递方向。

如果传递方向是D（direct），表示的含义如下：

当发生E类型的事件时，若事件主体带有标签L1，则为事件客体加上标签L2。说明主体的状态通过这个事件影响了客体的状态。

如果传递方向是R（reverse），表示的含义如下：

当发生E类型的事件时，若事件客体带有标签L2，则为事件主体加上标签L1。说明客体的状态通过这个事件影响了主体的状态。

举个例子，上图中的第1条规则PT1,FILE\_WRITE,FT1,D，表达的含义是，对于一次进程A写文件B的事件，若进程A带有PT1，则给文件B加上FT1的标签。过程如图4所示。

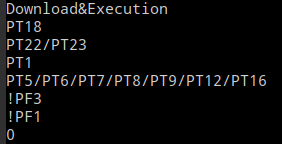
上图中的第6条规则PT3,FILE\_READ,FT1,R，表示对于一次进程A读文件B的事件，若文件B带有FT1，则给进程A加上PT1的标签。

规则中的/表示或的关系。

### 1.4.3 恶意状态

当某个进程带上了某些特殊种类的标签集合时，我们就认为该进程进入了恶意状态，并将其报告出来。判断一个进程是否进入恶意状态的规则写在judge.txt中。

下图展示了一部分judge.txt的内容：



开头的Download&Execution说明接下来的规则用于定义“下载并执行”这一恶意状态，结尾的0表示定义结束。中间的每一行代表一个标签，当某个进程含有所有这些标签时，就会被认为符合了“下载并执行“的定义，被报出来。规则中的/表示或的关系，!表示否的关系。

## 1.5 数据持久化

为了防止内存不断上涨，我们在内存中只保存当前存活的进程和文件，结束的进程和被删除的文件会被移出内存。但在进行溯源分析时，我们可能会用到已经结束的进程或被删除的文件。因此我们需要将数据存在硬盘中。目前我们的策略是当在内存中新建一个节点或关系时，同时在neo4j中也做同样的操作。但当从内存中删除节点或关系时，不去删除neo4j中的。我们只基于内存做检测，溯源分析完全基于neo4j。

# 2. 功能模块介绍

以下介绍基于http://118.126.94.181/magicshield/detector/blob/feature/refactor\_framework

部分内容参考http://118.126.94.181/weihao/LCSDetector/tree/E5\_windows\_linux

## 2.1 Main

**ApiLCSDetectorMain.java**

Online模式的入口。

**OfflineMain.java**

用于离线读取KafkaReverseConversion产生的数据。KafkaReverseConversion是用于从kafka读取数据并产生json的工具。在192.168.1.128 /home/weihao/KafkaReverseConversion可以获取。

**OfflineMainLocalJson.java**

用于离线读取Collector在json模式下产生的数据。因为历史原因，KafkaReverseConversion产生的数据格式和Collector在json模式下产生的数据的格式不一样，所以需要两个类来读。

## 2.2 KafkaProcessor

用于从kafka读取数据并产生detector可用的事件。

MessageReceiver用于接收，解压和反序列化数据，MessageProcessor负责根据format（详见1.2节）构建detector可用的事件，并按照事件类型的不同发送给不同的检测模块。

按照设计，一个MessageReceiver使用一个线程，并最多读10个topic的数据。一个MessageProcessor对应一个MessageReceiver，通过一个队列读来自其对应的MessageReceiver的数据。

ReceiverManager负责调控Receiver和Processor。

**ReceiverManager.java**

*addTopic():* 让detector开始读这些topic。查看当前是否有还有空位的MessageReceiver/Processor对，如果有，让这个有空位的Receiver/Processor对来处理这个新topic，没有的话，新建一对用于处理。

*requestStopTopic():* 让detector停止读这些topic。不能强制停止读取，只能请求其停止读取，这是因为，使用这个方法的场景是当collector停止时，要求detector停止读取这个collector的topic。在collector停止时，并不知道detector是否已经读完了这个topic上的所有数据。如果贸然停止，可能会遗漏数据。因此只是发一个停止请求，当detector读完了这个topic上的数据时才停止。

*resetClient():* 当某个client上线时，清空detector中所有关于它的数据。

**MessageReceiver.java**

*addTopic():* 开始读这些topic。

*requestStopTopic():* 将这些topic标记为应该停止。当某个被标记为应该停止的topic在最近MAX\_FAIL\_TIME次的poll中都没有读到数据时，判定其上的数据已经被读完，停止读topic。

*stopAndReport():* 执行停止操作，并报告daemon。

*updateSubscribedTopic():* 更新当前读取的topic。当新增topic或减少topic时调用。

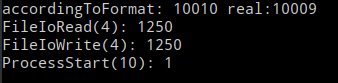
*run():* 永远循环，从当前订阅的topic上读取消息，解压，反序列化，放入队列等待对应的MessageProcessor读取。

**MessageProcessor.java**

*run():* 永远循环，从对应的MessageReceiver的队列中读取数据，并创建detector可用的事件，并进行分发。

*preCheck():* 对数据进行检测，判断根据format算出来的argument总数和实际收到的argument数量是否相同（见上文NOTE1）。若不同，会产生error\_report.txt文件。

下图是一个error\_report实例：



根据format算出来应该有10010个argument，但实际上只有10009个。

接下来列出了每种事件应该有的argument数量和这种事件出现的次数。

当我们假设collector发送的同一种类的事件的argument数量是不变的时，可以很容易猜出来导致format error的原因是collector在ProcessStart事件中少发了一个argument。

这是一个简单的例子，实际情况中可能发生更复杂的情况，调试就比较需要运气了。

*constructEventRecord():* 创建EventRecord对象，并分发给不同的检测模块。

*distributeRecord():* 分发EventRecord对象。

*processCallstack():* 处理callstack类型的事件。这一类型的事件并不携带callstack，这么叫是历史遗留问题。这些事件实际上在callstack字段携带PHF。比如当collector检测出pid为123的进程执行了RemoteDesktop操作时，就会发送一条Callstack事件，processId为123，callstack字段为RemoteDesktop。根据PHF的不同，我们创建不同的标签送入Detection Framework。

*processVisibleWindow():* 处理VisibleWindow类型的事件。这种事件表示某进程有可见窗口，窗口大小由left，right，top，bottom四个参数定义。当此事件的pid和tid都为0时，表示用户的屏幕大小。当窗口在屏幕范围外时，也会被认为不可见。提取出各个参数后，放入VisibleWindow模块进行检测。

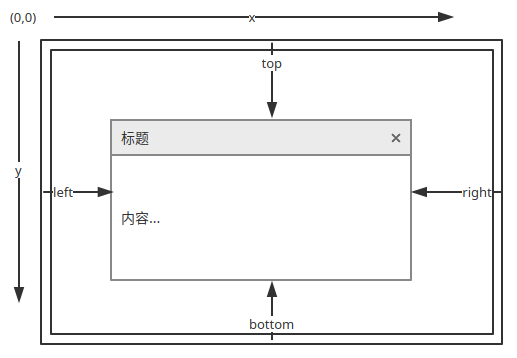


图5 可见窗口的参数含义

*processFileIoRead():* 处理FileIoRead事件。主要操作是将FILE\_READ事件发送给framework。

*processFileIoWrite():* 处理FileIoWrite事件。主要操作是将FILE\_WRITE事件发送给framework。其次，当某个文件以:zone.identifier结尾时，可以确定去除:zone.identifier部分的文件名对应的文件是被浏览器下载的。即若有C:\\Users\\weihao\\a.exe:zone.identifier，可以确定C:\\Users\\weihao\\a.exe是被下载的。创建一个标签将这件事告知framework。

*processFileIoRename():* 处理FileIoRename事件。这个事件表示重命名，参数中包含了旧文件名和新文件名。创建一个FILE\_RENAME事件发送给framework。

*processImageLoad():* 处理ImageLoad事件。这个事件表示进程加载文件。一般会加载exe或dll。参数中有一个Certificate字段，为0时表示加载的文件没有数字签名，为1时表示加载的文件有数字签名，为2时表示暂时不确定是否有数字签名。之所以有2这个状态，是因为collector判断数字签名比较耗时间，而detector由需要ImageLoad事件尽快发来，所以当collector端出现一个ImageLoad时，会先按2发，等判断出是否有数字签名后，再按照0或1发一次。

我们会先创建一个IMAGE\_LOAD事件发送给framework。当我们发现一个进程加载了自身的exe时，还会创建一个LOAD\_OWN\_EXE事件发送给framework。最后，将事件送入Certificate模块进行检测。

*processProcessStart():* 处理ProcessStart事件。这个事件表示父进程启动子进程。这个事件的argument含有以下字段：

* CommandLine: 进程启动的命令行。
* ImageFileName: 进程的exe的文件名。需要注意这里只有一个文件名，不包括完整路径。比如对于C:\\Users\\weihao\\a.exe的启动，这个字段只有a.exe。需要结合ImageLoad事件算出进程的完整名称。因为一个进程肯定会加载自身的exe，而ImageLoad事件中会带有exe的完整路径。若将两个事件匹配起来，就能得到完整的名称。详见2.6 ProcessTable.java
* ParentId: 父进程的pid
* ProcessId: 被启动进程的pid

先输入SensitiveProgram做检测，再将PROCESS\_START事件输入framework。最后，如果被启动的是一个powershell，则需要用PowerShell模块检测其命令行。

*processProcessEnd():* 处理ProcessEnd事件，主要操作是创建一个PROCESS\_END事件发送给framework。

*processNetworkConnect():* 处理TcpIpConnectIPV4和UdpIpConnectIPV4事件。主要操作是送入NetworkActivity模块进行检测。这些网络事件中包括以下字段：

* sport：本地端口
* saddr：本地ip地址
* dport：远程端口
* daddr：远程ip地址

有些时候数据中s和d会互换。这种情况可以忽略。

*processNetworkClose():* 处理TcpIpDisconnectIPV4和UdpIpDisconnectIPV4事件。主要操作是送入NetworkActivity模块进行检测。

*processNetworkSend():* 处理TcpIpSendIPV4和UdpIpSendIPV4事件。主要操作是送入NetworkActivity模块进行检测，再创建一个NETWORK\_SEND事件发送给framework。

*processNetworkRecv():* 处理TcpIpRecvIPV4和UdpIpRecvIPV4事件。主要操作是送入NetworkActivity模块进行检测，再创建一个NETWORK\_RECV事件发送给framework。

*processThreadStart():* 处理ThreadStart事件，这个事件表示线程的创建。事件的processId表示创建者的pid，事件的threadId表示创建者的tid，参数中的ProcessId表示被创建的线程所属的进程的pid，参数中的ThreadId表示被创建的线程的tid。主要操作是将其送入ProcessInjection进行检测。

**TopicKey.java**

把topicName，clientId等信息打包。

## 2.3 ETWproto

由protobuf自动生成的代码，用于反序列化protobuf消息。不要修改。

## 2.4 EventRecord

**EtwEventType.java**

枚举，包含了所有collector会发送过来的事件类型。当collector要增加新的事件类型时，在此添加。

**EventFormat.java**

从format.txt中读取format信息。

详见1.2节。

**EventRecord.java**

即1.2节中提到的事件。

**ProtobufParsedMessage.java**

将Protobuf反序列化后产生的数组（即图3中的数组，实际是一个EventRecords对象）和clientId（客户机器的ID）打包。

## 2.5 ExtraDetectionFactors

包含各个检测模块。里面的很多类其实没什么用，可以考虑重构时合并。

**Certificate.java**

用于检测数字签名。若某文件没有数字签名，创建一个标签FT2，送往framework。若某个进程加载了自己的exe，且该exe没有签名，创建一个标签PT18，送往framework。

**Macro.java**

用于检测宏。若某office文件含有宏，创建一个标签FT4，送往framework。

**NetworkActivity.java**

用于检测网络活动。当某个进程有网络活动时，创建一个标签PT1送往framework。

**PowerShell.java**

用于检测powershell。从signature文件中读取恶意powershell的pattern，与送入的CommandLine进行匹配，若匹配上，创建一个标签送往framework（目前没用上）。

**ProcessInjection.java**

用于检测进程注入。进程注入指某个恶意进程将自己的代码放到另一个正常进程中执行，以逃避检测。我们检测以下这种形式的进程注入：恶意进程A在正常进程B中创建了一个线程，用于执行它的恶意代码。这种注入会造成两个现象：

* 收到进程A的ProcessInjection的PHF
* 收到一个ThreadStart事件，事件的processId为A的pid，事件的threadId为A的某个线程的tid，argument中的ProcessId为B的pid，argument中的ThreadId为B中恶意线程的tid。

我们需要结合这两个事件来检测注入。

需要注意，我们对注入的检测是线程级别的，

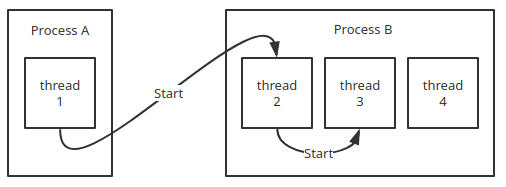


图6 线程级别的注入检测

在图6中，进程A的线程1注入了进程B，启动了线程2，线程2启动了线程3。我们会将线程2和线程3的PHF都归到进程A上。但对于线程4的PHF，它仍然会被算在进程B上。

但我们对网络的归类是进程级别的，即所有进程B的网络连接都被归到了进程A上，即使这个连接是由线程4执行的。

具体的检测方法，是每当有ThreadStart和ProcessInjection的PHF时，都通过调用 *addThreadStart()* 和 *addPossibleInjection()* 储存起来，并调用 *checkIfMatch()* 按照上文提到的两个现象尝试进行匹配。当配对成功时，就算检测到了一个注入。

*getRealProcess():* 当接收到某个进程的某个线程的PHF事件时，调用这个方法，看该线程是否被注入，若被注入，则返回注入者，也就是将这个PHF归到注入者上。

*guessRealProcess():* 将被注入进程的网络活动归到注入者上。

**SensitiveFile.java**

检测进程是否访问敏感文件。敏感文件在signature文件中被定义，在这里对所有的FileIoRead/FileIoWrite事件进行匹配，若匹配上，说明该进程访问了敏感文件，为该文件创建一个标签FT5并送往framework。并不需要手工为进程创建PT2标签，framework会根据规则自动创建。

**SensitiveProgram.java**

检测进程是否执行敏感命令。敏感命令在signature文件中被定义，这里对所有的ProcessStart事件进行匹配，若某个进程启动了两个以上的敏感命令，为其创建PT12标签并送往framework。

如果被启动的是powershell或python等脚本，需要特殊处理，创建一个START\_SCRIPT事件并送往framework。这是因为powershell和python自身是正常程序，有数字签名，是可信的，但可以用来执行不可信的脚本，达成不可信的意图，因此需要单独处理。

**VisibleWindow.java**

检测程序是否有可见窗口。若窗口的长或宽小于等于2，判定为太小而不可见；若窗口在屏幕外，如下图的情况：

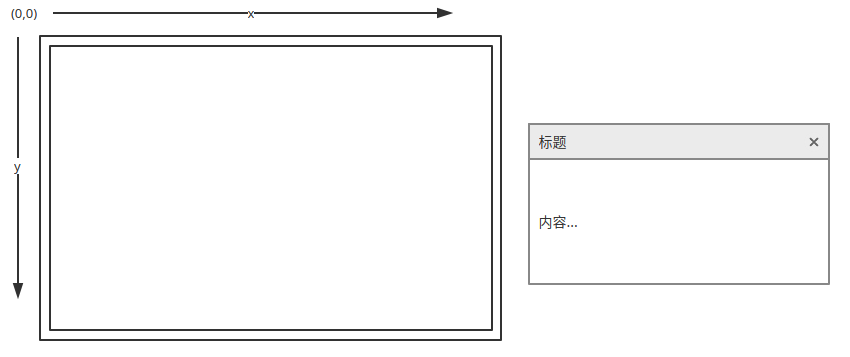


图7 屏幕范围外的窗口

会被认为不可见。若某个进程有可见窗口，为其创建一个PF3标签并送往framework。

目前这部分的判断规则还不稳定，可能会有变化。

*newProcessStarted():* 详见2.6 ProcessTable.java

## 2.6 Process

**Process.java & ProcessTable.java**

Process.java储存一个进程的基本信息，包括以下字段：

* clientId：该进程来自的客户机器的id
* uuid：因为pid会出现重复，因此需要uuid来做为唯一标识
* processId：进程id
* startTime：进程启动时间，默认为0
* endTime：进程结束时间，默认为0
* processName：进程完整名称

### 计算ProcessName

在2.3 MessageProcessor processProcessStart()中提到，ProcessStart事件只含有短文件名，ImageLoad事件含有完整文件名，因此需要将两者结合才能得到进程的完整名称（一个进程ImageLoad不会只load自己的exe，也可能load其它exe，因此不能只依靠ImageLoad事件）。

举个例子，当有以下事件时：

pid: 123, ProcessStart, argument: {ProcessId = 456, ParentId = 123, FileName = a.exe}

pid: 456, ImageLoad, argument: {FileName = C:\\Users\\weihao\\b.exe}

pid: 456, ImageLoad, argument: {FileName = C:\\Users\\weihao\\c.exe}

pid: 456, ImageLoad, argument: {FileName = C:\\Users\\weihao\\a.exe}

通过匹配ProcessStart的FileName字段和ImageLoad的FileName字段，可以得知完整进程名为C:\\Users\\weihao\\a.exe。

### 计算uuid

首先，因为在同一时间不会有两个相同pid的进程，所以当给定一个pid和一个timestamp时，可以唯一确定一个进程。

然后我们发现了这样一个现象：**现象1：某个进程的ProcessStart事件具有最早的时间戳，ProcessEnd事件具有最晚的时间戳，其它事件的时间戳在这两个时间戳划定的范围内。**我们可以利用这一现象来计算uuid。

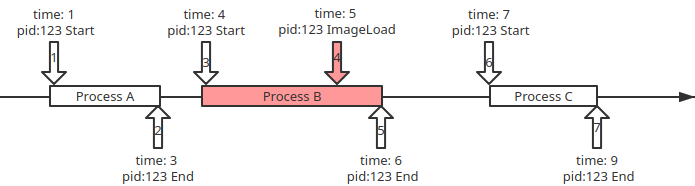


图8 给定pid和时间戳 确定唯一进程

ProcessTable中有一个processId2ProcessList的map，key为processId，value为一个Process的list，按照时间顺序储存着所有具有这个pid的Process。下图是一个例子：

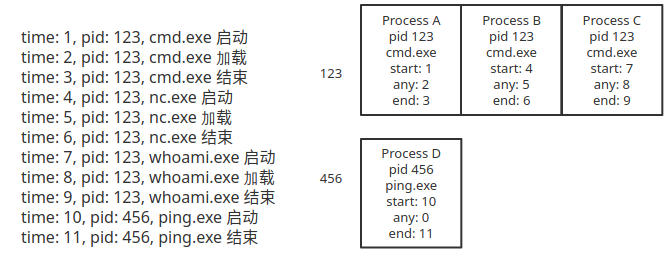


图9 processId2ProcessList的结构

ProcessTable只会接收以下三类事件：

* ProcessStart，更新Process的startTime
* ProcessEnd，更新Process的endTime
* ImageLoad，更新Process的anyMiddleTime

因为Collector发送的数据并非严格按照时序，即有可能时间戳晚的事件可能比时间戳早的事件先来，所以一个进程的ImageLoad事件可能在其ProcessStart之前到达，即图8中的4号箭头可能比3号箭头先到达。但因为windows会隔一段时间再重用pid，因此不可能图8中的4号箭头比2号箭头先到达，更不可能图8中的5号箭头比2号箭头先到达。

以图8为例，若所有箭头都按照时序到来，则建立图9中的processId2ProcessList没有任何问题。但若发送了时序混乱，比如箭头按照1,2,4,3,5,6,7的顺序到来，应该这样处理：

假设目前已经来了1号箭头和2号箭头，于是我们成功建立了Process A。接下来来了4号箭头，观察到4号箭头的时间戳5比Process A的EndTime 3更晚，因此4号箭头肯定不属于Process A，因此创建一个Process B。其startTime为默认的0，endTime为默认的0，anyMiddleTime为4号箭头的时间戳5。

接下来来了3号箭头，观察到它的时间戳4比Process A的endTime 3更晚，因此不属于Process A，继续往后搜索，发现刚刚建立的Process B，其startTime为0，anyMiddleTime 5比3号箭头的时间戳4大，因此可以判断3号箭头属于Process B，于是更新Process B的startTime为4。

为每一个创建的Process赋予一个单调递增的int，作为其uuid。

上文提到的现象1有一个例外，即某个进程的VisibleWindow事件的时间戳可能比其ProcessStart事件的时间戳早。因此有可能某个VisibleWindow事件找不到对应的进程。这种情况下，VisibleWindow.java会先将这个pid缓存起来，当ProcessTable得知某个进程启动时，通过VisibleWindow的newProcessStarted方法告知VisibleWindow。

## 2.7 Framework

**Label/LabelType.java**

一个枚举，包含了所有的标签种类。其中以P开头的是属于进程的标签，以F开头的是属于文件的标签。某些P标签比较特殊，叫做PHF标签，说明带有该标签的节点执行了Potential Harmful Function。当某个节点带有PHF标签时，我们才会对其进行检测。

**Label/LabelForCS.java**

外界向Framework传递信息，让Framework中的某个Node带上某个标签的类。

包含以下内容：

* LabelType：说明需要带上哪种标签
* uuid：若需要ProcessNode带上标签，则这个域设为该ProcessNode的uuid
* stringField：若需要FileNode带上标签，则这个域设为该FileNode的FileName

等等

**Event/EventType.java**

一个枚举，包括了所有的事件种类。

**Event/EventForCS.java**

外界向Framework传递信息，说明发生了某个事件的类。

包含了以下内容：

EventType：事件种类

Process：该事件的主体的进程，需要注意若为ProcessStart事件，该域表示被启动者，而非启动者

timestamp：事件发生的时间

parentUuid：若该事件为ProcessStart事件，则该域为启动者的uuid

string1：若该事件为FileRead/FileWrite/FileDelete，则该域为被读/写/删除的文件的文件名；若该事件为FileRename，则该域为被改名文件的旧文件名

string2：若该事件为FileRename，则该域为被改名文件的新文件名

**Event/NetworkEventForCS.java**

外界向Framework传递信息，说明发送了网络传输事件的类。事件类型可能为NETWORK\_SEND, NETWORK\_RECV, NETWORK\_CLOSE

继承了EventForCS，增加了以下四个域：

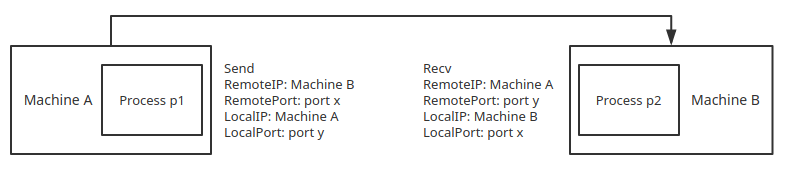
saddr：本地IP地址

daddr：远程IP地址

sport：本地端口

dport：远程端口

这种事件用来检测跨机器数据流。像下图这种情况，就会认为Process p1和Process p2之间有联系。



*getIntranetSocket()*：获取这个事件对应的IntranetSocket，见2.9 Framework/IntranetSocket.java

**ISignalForCS.java**

一个接口，被EventForCS和LabelForCS实现。可以用作对输入的EventForCS和LabelForCS进行排序。因为现在的Collector发送数据并非严格按照时序，但我们的检测框架要求数据严格按照时序。在不排序的情况下可以做检测，且效果很好。但如果未来有需求，还是需要按照timestamp来排序。具体代码参考http://118.126.94.181/weihao/LCSDetector/tree/E5\_windows\_linux/src/main/java/NU/ETWRealTimeDetector/CrossMachine

SingleMachineSignalPool建了一个大小为5000的ISignalForCS的最小堆，从而以5000的窗口来排序。因为collector虽然会时序混乱，但不会太混乱，实验表明5000的窗口足以实现比较好的排序效果。这一步实现了机器内部的排序，但如果需要对所有机器的数据进行排序，则需要MultiMachineSignalPool。

MultiMachineSignalPool实现了对所有机器的排序。若有N台机器，则建立一个大小为N的最小堆，把来自每个机器的当前最小的ISignalForCS放入该堆再取出最小的。

SignalMachinePool和MultiMachinePool与之类似，但排序的是EventRecord而非ISignalForCS。对应的，窗口需要设大一点，实验表明30000的窗口可以实现比较好的排序效果。

**Grammar**

包含对label.txt，trans.txt和judge.txt的解析等，基本不言自明。

**Node/Node.java**

是ProcessNode和FileNode的父类。储存一个BitSet用来表示该节点所带的状态。若BitSet第一位为1，则说明该节点带有第1个标签；若BitSet第二位为0，则说明该节点不带有第2个标签。所谓第一个标签，指的是LabelType中ordinal为1的标签。BitSet的大小为所有Label的数量，通过LabelType.getSize()获取。

needsJudge表示该节点是否需要被检测。若该节点不含任何PHF标签，则该值为false；若该节点含有PHF标签，则该值为true。

needsJudgeNow表示该节点当前是否需要被检测。若该节点needsJudge，且被加了一个新的标签，则该值被设为true；若加了一个标签，但该标签之前已经有了，则该值为false。

只有当一个节点needsJudgeNow时，我们才会检测它。检测完后，通过turnOffJudgeNow来将needsJudgeNow设为false。

**Node/ProcessNode.java**

一个ProcessNode表示一个进程。包含其对应的Process，有一个对其父进程节点的引用parentNode，有一个List of childNodes。含有两个BitSet，用来表示它因为哪些PHF被报告过了。目前的逻辑是，若某个进程节点已经被报过一遍了，那如果它新增了一个PHF标签，就需要再被报一遍。如果BitSet的第一位为1，说明第一个PHF已经被报过了。如果BitSet第二位为0，说明第二个PHF没有被报过。PHF排第几个，通过LabelType.getPhfIndex()来获取。BitSet的大小为所有PHF标签的数量，通过LabelType.getPhfSize()获取。

ProcessNode通过其Process的uuid来做唯一标识。

**Node/FileNode.java**

一个FileNode表示一个文件和其uuid。FileNode的uuid从-1开始递减，与ProcessNode的uuid从0开始递增区分开来。所以若某Node的uuid大于0，则为ProcessNode；uuid小于0，则为FileNode。

**DetectionFramework.java**

* judgeMode2AlertLevel：framework会报出不同等级的报警，而在与webportal交互时，需要将这些报警等级转换成数字发过去。
* monitorJudgeMode2AlertLevel：framework会报出不同等级的监控报警，而在与webportal交互时，需要将这些监控报警等级转换成数字发过去。

**报警指我们确定的攻击，模式在judge.txt中定义；监控报警指该进程只满足一部分攻击模式，但为了让用户感觉到我们在进行监控，仍然将其报出来。模式在monitor\_judge.txt中定义。**

* fileMap：以filename为key，对应的FileNode为value的表
* processMap：以uuid为key，对应的ProcessNode为value的表
* processRoot：假设存在的，最老的ProcessNode，作为所有没有parent的ProcessNode的父节点
* dyingProcessNodes：**在收到某个进程的ProcessEnd后，应该将其从内存中删除。但往往在ProcessEnd之后仍然会收到属于该进程的一些事件。如果在ProcessEnd后直接移除，收到后续事件后就会再创建一个新的ProcessNode，且该ProcessNode永远不会被移除。因此需要对ProcessEnd延迟处理。我们的实现是，当收到ProcessEnd时，将该ProcessNode放入一个长度为5的队列，当期被移除队列时，才将其从内存中删除。这样基本上可以避免上述的情况发生。**
* dyingFileNodes：在收到某个文件的FileIoDelete后，应该将其从内存中移除。后续逻辑与Process一致，略。
* dyingNodesLimit：用于设置dyingProcessNodes和dyingFileNodes的大小。

*DetectionFramework()*：读取各种grammar，往ProcessTable中加入processRoot，创建用于写结果的目录。

*insertNewNodeUnderRoot()*：把某个ProcessNode设为processRoot的子进程。

*getProcessNodeByEvent()*：获取某个EventForCS的主体进程对应的ProcessNode，如果没有找到，则创建一个新的ProcessNode并用insertNewNodeUnderRoot()挂到processRoot下。

*getProcessNodeByUuid()*：通过uuid获取对应的ProcessNode，如果没有找到，则创建一个新的，挂到processRoot下。

*getFileNodeBySignal()*：通过ISignalForCS获取对应的FileNode。如果是EventForCS，获取到的是客体的FileNode；如果是LabelForCS，获取到的是要加上label的FileNode。**如果在fileMap中没有找到，说明这个file是第一次来，也说明它上面不会有任何标签。而如果这是一个针对该文件的读操作或改名操作，因为该文件上没有标签，所有读操作不会造成任何标签传递，所以我们完全不需要储存这个File。**

*addEvent()*：外界调用该方法，向framework中添加事件。对于不同类型的事件，调用不同类型的处理方法来处理。

*processProcessStartEvent()*：处理PROCESS\_START事件。将childProcess的父进程设为parentProcess，往parentProcess的子进程中添加childProcess。

*processProcessEndEvent()*：处理PROCESS\_END事件。将该ProcessNode放入dyingProcessNodes。如果当前队列大小超过限制，则用removeProcess()移除队列中最早的ProcessNode。

*removeProcessNode()*：将ProcessNode从processMap中移除，并删除其所涉及到的所有父子关系。

*processFileEvent()*：处理FILE\_READ/FILE\_WRITE/FILE\_DELETE/FILE\_RENAME事件。如果是FILE\_DELETE，则将FileNode放入dyingFileNodes。若队列长度超过限制，则调用removeFileNode将最先进入队列的FileNode移除。如果是FileIoRename，则对fileMap和FileNode进行修改，同时通知neo4j进行修改。如果是其它的，则调用transLabel进行标签传递。

*processLoadOwnExeEvent()*：处理LOAD\_OWN\_EXE事件。该事件指某进程ImageLoad了自己的exe。这是通过对比被load的exe和processName是否相等判断的。处理方法是进行传递标签。

*processInjectionOrScriptEvent()*：处理INJECTION/START\_SCRIPT事件。处理方法是传递标签。

*addLabel()*：外界调用该方法，向framework中的某个Node添加某个Label。方法是调用该node的addLabel方法，且通知neo4j。如果添加标签后，该节点needsJudgeNow，则调用judgeNode进行judge，然后turnOffJudgeNow。

*transLabel()*：用于传递标签。若要求的传递方向为正向或双向，则获取所有符合主体标签和事件类型的传递规则，依次调用performTransfer()进行传递；若要求的传递方向为反向或双向，则获取所有符合客体标签和事件类型的传递规则，依次调用performTransfer()进行传递。

*performTransfer()*：实际传递标签的方法。如果是正向传递，则在客体上添加标签，通知neo4j，然后判断客体是否需要judge；如果是反向传递，则在主体上添加标签，通知neo4j，然后判断主体是否需要judge。

*judgeNode()*：使用judgeGrammar的judge方法判断某节点是否符合judge.txt或monitor\_judge.txt定义的恶意状态。对于judge.txt里面定义的一组label，如果当前ProcessNode带的所有label是其超集，则认为符合该模式，返回该模式的名称。如果不符合任何一个模式，返回空字符串。如果返回的不是空字符串，则向webportal报告。且更新PhfHasBeenReported，防止下一次因为相同PHF而报警。

*outputResult()*：将检测结果写到硬盘。

*reportToDaemon()*：将检测结果汇报到webportal。

## 2.8 Neo4j

处理关于neo4j的任务。任务包括创建节点，创建关系，修改节点等。每个任务作为一个对象放入队列，异步执行。

**Task/BaseTask.java**

所有任务的父类。

**Task/AddFileNodeTask.java**

在neo4j中添加一个文件节点。节点标签为FNode，储存uuid和filename。

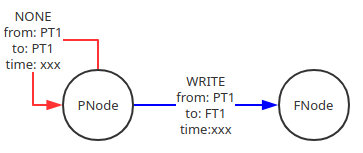
**Task/AddProcessNodeTask.java**

在neo4j中添加一个进程节点。节点标签为PNode，储存uuid, processid, startTime, endTime和processName。

**Task/AddLabelTask.java**

用于添加标签。若该标签是直接产生的，则在需要添加的节点上创建一个指向自己的Relationship，relationship的name是NONE，from和to都是这个标签，time为创建时间，如图中的红线所示。

若该标签是通过传递规则产生的，则创建从源节点到本节点的一条Relationship，Relationship的name是产生标签的事件名，from是源标签名，to是此标签名，time为创建时间，如图中的蓝线所示。



**所以要获取一个节点的所有标签时，可以获取它所有INCOME的Relationship，并获取它们的to字段（但请注意AddNodeRelationTask中的加粗字体）。**

**Task/AddNodeRelationTask.java**

我们会通过这个task在neo4j中储存父子进程关系，即创建父进程到子进程的一条Relationship，name为START，time为创建时间。所以需要注意，**并非所有的INCOME的Relationship都有to字段，需要做过滤。**

**Task/ChangeNameTask.java**

当某个文件被改名是，将其旧名存入neo4j。也就是存入对应文件节点的original字段（多个旧名间用\n分割），并将节点的name改成新名字。

**Task/DeleteNodeRelationTask.java**

删除某两个节点之间的关系，一般发生在某个被挂在processRoot下的进程节点找到了真正的parentProcess时，删除该进程和processRoot之间的父子关系。

**Task/UpdateProcessNodeTask.java**

更新某个进程节点的信息，一般发生在该进程的ProcessEnd来到后，更新endTime；或获取到该进程的processName后，更新ProcessName等。

**Neo4jClient.java**

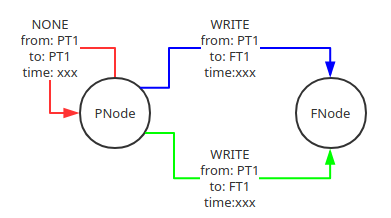
单例。外界通过调用它的方法来实现写neo4j的操作。

queue：一个队列，里面存着各种Task，当到达一定数量后，一起放在一个transaction里面执行。

cacheSize：规定队列的大小，目前为10000。

queueSize：获取ConcurrentLinkedQueue的size比较耗时，因此使用一个AtomicInteger来记录size。

set：需要对AddLabelTask进行去重。考虑下图的情况：



PNode带有PT1，随后两次写FNode。这就会产生两个Relationship，如图中的蓝线和绿线所示。我们需要尽可能防止这种情况发生，因此需要去重。我们的去重方法是，对于放入queue中的task，同时还放入一个set。而两个AddLabelTask，如果只有time不一样，则认为是相同的。每新来一个AddLabelTask，先检测set中是否有相同的，如果有，则直接丢掉。我们在大小为10000的窗口下进行这种去重，效果较为良好。扩大窗口大小能实现更好的去重效果，而代价是写入neo4j的延迟会更大。

*run()*：死循环，不断检查当前queue里面的task数量是否达到cacheSize，如果达到，则进行flush；如果没有达到，计数器加1，并休眠一段时间。当计数器到达某个阈值时，强制flush。

*flush()*：获取队列里面一定数量的task予以执行。

**2.9 CrossMachine**

检测跨机器的连接，见2.7 Framework/Event/NetworkEventForCS.java

**IntranetSocket.java**

用于表示一台机器上的一个socket，包括以下字段：

* remoteAddress：远程IP地址
* localAddress：本地IP地址
* remotePort：远程端口
* localPort：本地端口
* clientId：机器名
* eventType：Send/Receive（将接收的和发送的视为两个socket）

*getName()*：下表中的四个Socket都将有一样的名字。

|  |  |  |  |
| --- | --- | --- | --- |
| Socket 1 | Socket 2 | Socket 3 | Socket 4 |
| remoteAddress: XXXX | remoteAddress: XXXX | remoteAddress: YYYY | remoteAddress: YYYY |
| remotePort: 123 | remotePort: 123 | remotePort: 456 | remotePort: 456 |
| localAddress: YYYY | localAddress: YYYY | localAddress: XXXX | localAddress: XXXX |
| localPort: 456 | localPort: 456 | localPort: 123 | localPort: 123 |
| SEND | RECV | SEND | RECV |

*matches()*：若两个socket在不同的机器上，getName()一样，且一个发送一个接收，则两个socket匹配。

**ProcessNetworkEvent**

代码见[http://118.126.94.181/weihao/LCSDetector/blob/E5\_windows\_linux/src/main/java/NU/ETWRealTimeDetector/Framework/DetectionFramework.java#L295](http://118.126.94.181/weihao/LCSDetector/blob/E5_windows_linux/src/main/java/NU/ETWRealTimeDetector/Framework/DetectionFramework.java" \l "L295)

在Framework中，对NetworkEventForCS进行处理，从中发现跨机器的连接。

首先在DetectionFramework中加两个域：

* socketName2Socket：以socket的getName()为key，以所有拥有这个getName()的socket的集合为value的map。
* socket2ProcessNodes：以socket为key，以所有拥有这个socket的ProcessNode的集合为value的map。当某个process为某个NetworkEventForCS的主体时，我们说这个Process拥有这个NetworkEventForCS对应的socket。

对于输入的NetworkEventForCS，若类型为SEND或RECV，应尝试进行配对；若为CLOSE，应将对应的socket从内存中移除。

先看SEND或RECV：设该NetworkEventForCS的主体为Process A，getIntranetSocket()得到的是Socket A。首先往socket2ProcessNodes里面加入ProcessNode A，然后从socketName2Socket中拿出所有和socket A同名的socket，依次与socket A比较，看是否matches。如果没有match的，返回；如果有，设对应的socket为socket B，从socket2ProcessNodes中拿出socket A对应的ProcessNodes，设为ProcessNodes X，从socket2ProcessNodes中拿出socket B对应的ProcessNodes，设为ProcessNodes Y。我们认为ProcessNodes X和ProcessNodes Y有交互，因此在它们之间进行标签传递。最后将socket A加入socketName2Socket。

再看CLOSE：将这个socket从socketName2Socket和socket2ProcessNodes中移除。

## 2.10 Illustrator

代码见<http://118.126.94.181/weihao/Illustrator>

基于neo4j里面的内容画攻击链。

**node/BaseNode.java**

node中的类，是对neo4j中的node的复制。我们先对neo4j中的涉事节点复制成内存中的node，然后基于内存中的node来画图。

BaseNode是ProcessNode和FileNode的父类，包含uuid和edges。edges为其所有出/入边（见edge/BaseEdge.java)的Set。

**node/FileNode.java**

文件节点，储存fileName和originalName。

**node/ProcessNode.java**

进程节点，储存processName，startTime，endTime，processId。

**edge/BaseEdge.java**

edge中的类，表示从某个节点的出边或入边。

BaseEdge是IncomingEdge和OutgoingEdge的父类，包括以下域：

remoteUuid：边的另一边的节点的uuid

eventType：边表示的事件类型

localLabelType：这条边表示的标签传递的属于本节点的label类型

remoteLabelType：这条边表示的标签传递的属于另一方节点的label类型

timestamp：时间戳

但在比较BaseEdge时，会忽略掉timestamp。

**edge/IncomingEdge.java**

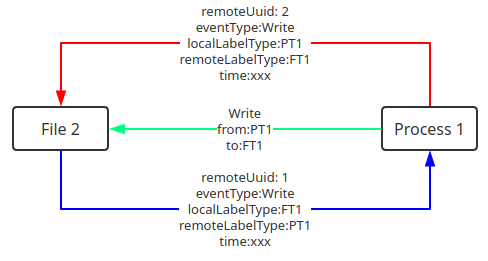
某节点的入边，继承BaseEdge，没有修改。

**edge/OutgoingEdge.java**

某节点的出边，继承BaseEdge，没有修改。

下图中，绿线是neo4j中的Relationship。

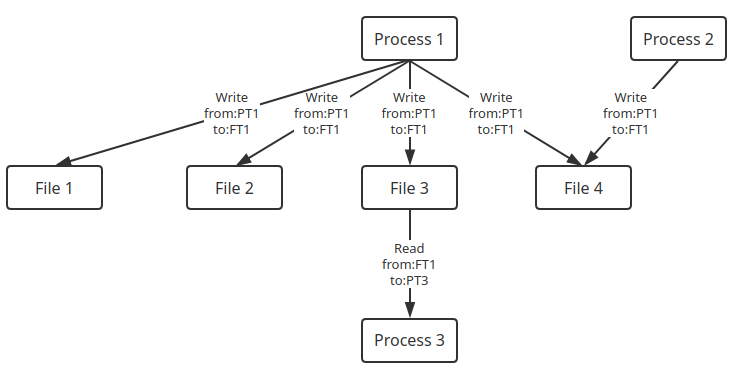
红线是Process1上存的出边，蓝线是File2上存的出边。



在节点上储存其所有的出边和入边，是为了找出所有等位节点。等位节点在NodeWrapper中介绍。

**NodeWrapper.java**

我们会对所有等位的文件节点进行合并。



若某两个节点带有的出边和入边完全相同，则两节点等位。

如上图所示，File1和File2等位，因为与它们相连的所有边，除了timestamp都相同。File3和它们不等位，因为它多了一条出边；File4和它们不等位，因为它多了一条入边。

NodeWrapper包含了一个List<BaseNode>。我们会把所有的ProcessNode或FileNode都放入NodeWrapper，但有的NodeWrapper里面只含有一个Node，有的里面含有多个等位的Node。

**Link.java**

表达将被画出来的图中的一条边。包含fromUuid（出发节点的uuid），toUuid（到达节点的uuid），eventType（事件类型）。

**Common.java**

含有一些公用的函数，比如constructNodeWrapper()用来将NodeWrapper转化成dot格式或json格式，constructProcess()用来将ProcessNode转化成dot格式或json格式，constructLink()用来将边转化成dot或json格式。

我们会画两种类型的图，在useLegacy时画dot类型的图，否则画json格式的图。dot格式的可以用命令

sudo apt install graphviz

dot -Tpdf xxx.dot > a.pdf

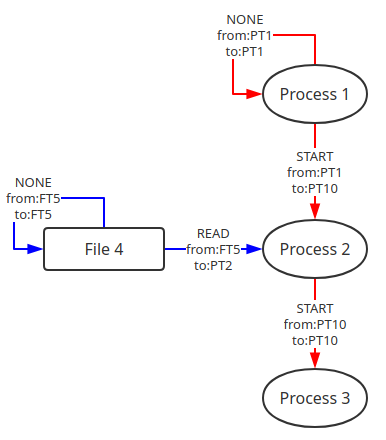
来生成pdf格式的图。

json格式的图传送给webportal，由webportal展示。

**DrawAttackGraph.java**

画某个节点的攻击图，也就是从该节点出发，追踪其所有的标签的源头。

方法是在neo4j中找到指定节点，依次从它的每个标签开始进行深度优先遍历。



如图所示，若我们要追踪Process3的PT10标签，可以追查到Process2的PT10标签，再可以追溯到Process1的PT1标签，即图中的红线。虽然Process2有PT2，可以追溯到File4，即图中的蓝线，但我们在追踪Process3的PT10时，不需要追踪蓝线。

可以看出，当我们在Process2上进行追踪时，并不会追踪Process2带有的其它无关标签。所以在某个节点进行追踪时，需要先限定需要追踪的标签集合。

*draw()*：得到出发的节点后，列举其所有的INCOMING的Relationship，排除掉不带to字段的，剩下的表示该节点带有的标签。建立一个needTrackLabels集合，把这些标签添加进去，说明这些标签需要被追踪。

然后遍历这些Relationship，从Relationship的另一端点出发，追踪这个Relationship的from标签。